

RL-TR-97-141
Final Technical Report
October 1997



HIGH PERFORMANCE COMPUTING ENVIRONMENTS

Software Options, Inc.

Sponsored by
Advanced Research Projects Agency
ARPA Order No. A183

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19980217 499

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

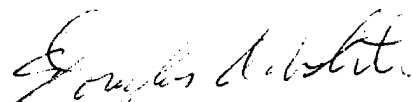
Rome Laboratory
Air Force Materiel Command
Rome, New York

DTIC QUALITY INSPECTED

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-141 has been reviewed and is approved for publication.

APPROVED:



DOUGLAS A. WHITE
Project Engineer

FOR THE DIRECTOR:



JOHN S. GRANIERO, Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CB, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

HIGH PERFORMANCE COMPUTING ENVIRONMENTS

Contractor: Software Options, Inc.
Contract Number: F30602-93-C-0185
Effective Date of Contract: 30 July 1993
Contract Expiration Date: 30 June 1996
Program Code Number: 5E20
Short Title of Work: High Performance Computing Environments
Period of Work Covered: Jul 93 – Jun 96

Principal Investigator: Michael Karr
Phone: (617) 497-5054
RL Project Engineer: Douglas White
Phone: (315) 330-2129

Approved for public release; distribution unlimited.

This research was supported by the Advanced Research Projects
Agency of the Department of Defense and was monitored by
Douglas White, RL/C3CB, 525 Brooks Rd, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE Oct 97	3. REPORT TYPE AND DATES COVERED Final Jul 93 - Jun 96	
4. TITLE AND SUBTITLE HIGH PERFORMANCE COMPUTING ENVIRONMENTS			5. FUNDING NUMBERS C - F30602-93-C-0185 PE - 62301E PR - A183 TA - 00 WU- 01	
6. AUTHOR(S) Judy Townley and Michael Karr				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Options, Inc. 22 Hilliard St. Cambridge, MA 02138			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Advanced Research Projects Agency 3701 Fairfax Drive Arlington, VA 22203-1714			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-97-141	
Rome Laboratory/C3CB 525 Brooks Rd Rome NY 13441-4505				
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Douglas A. White, C3CB, 315-330-2129				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE NA	
13. ABSTRACT (Maximum 200 words) This report describes research and development undertaken to develop an environment, consisting of compiler and debugger, for developing software for high performance computers that is able to exploit information about the execution of a program. In other words, this environment does more than merely optimize a program, it optimizes the program's performance on a given set of "typical" inputs.				
14. SUBJECT TERMS Software development environments, parallel computer, compilation, debugging			15. NUMBER OF PAGES 24	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Contents

1	Objectives and Goals	1
2	Approach	1
3	Distribution	2
3.1	General Information	2
3.2	When to Report a Bug	3
3.3	For the Experienced User	5
3.4	Pragmatics	6
3.5	Caveats	7
3.6	For GCC/GDB Cognoscenti Only	8

1 Objectives and Goals

We begin with a quote from the section of the same name in the Statement of Work for the Option:

The objective of this effort is to develop a system for debugging highly optimized code. We propose to develop techniques whereby *a user can debug an optimized program and yet be oblivious to the optimizations* and at the same time *the debugging technology places no constraints on allowable optimizations*. The work will develop a principled approach to the construction of such a system. It will further demonstrate the validity of these principles and the practicality of such a system by an implementation in the context of a state-of-the-art compiler/debugger pair.

We have succeeded in meeting these objectives. This report will provide an overview of what we have done and how we have done it. The reader should keep in mind that the effort in this project went into constructing an innovative, practical system, not into this report. Even the user documentation is minimal because one of the objectives stated above is that the user be oblivious to the capabilities we have provided.

2 Approach

We proposed to base our effort on the Gnu compiler/debugger pair of GCC and GDB. GCC is a state-of-the-art compiler that incorporates all the classical optimizations, and GDB is a better-than-most Unix debugger. GCC, even before our modifications, was capable of compiling for debugging (the `-g` switch) independent of the optimization level. (Many compilers will compile for debugging only when not optimizing.) However, the behavior of the debugger on programs compiled with optimization can be confusing and misleading, and will sometimes not provide certain capabilities to the user. Our goal here was to remedy this confusing, misleading, and missing behavior. The fact that our goal was to get GDB to behave “normally” is the reason for minimal user documentation of our improvements---GDB already has user documentation; our work simply makes GDB adhere closer to that documentation.

Our plan was to consider two or three kinds of optimizations. Our resources were sufficient for two, and we worked on the two optimizations in the order of priority that we originally suggested: inlining and register allocation. The inline optimization is important because it both is one of the most important in speeding up programs (particularly those written in C++) and most hinders ones ability to debug (ordinarily, it is not possible to set a breakpoint in an inlined function). After inlining, register allocation is the most troublesome with respect to debugging because it can hide the true values of variables.

We proposed that part of our effort would be spend in restructuring GDB into a “top” part providing the interface and a “bottom” part providing the abstraction of a machine execution. Once we came into close contact with the reality of GDB, we realized that this part of the proposal was

naive. In some ways, GDB is in need of such a restructuring. On the other hand, the effort would be massive. This is not so much a criticism of GDB as it is paying respect to the fact that GDB can debug programs on more target architectures, running on more hosts, using more symbol table formats, providing more features, than any other debugger. A debugger necessarily lives close to the operating system and close the hardware, and only by examining the code of GDB can one fully appreciate the degree to which this entails living dangerously---it is full of work-arounds because of operating system bugs, compiler bugs, hardware quirks, and the like.

So instead of restructuring GDB, we took the opposite tack and left the structure of GDB as invariant as possible, inserting all our changes under conditional compilation switches (bracketed by `#ifdef/#endif` constructs). We believe this will turn out to be a more pragmatic approach, because the changes can then be distributed as part of standard GDB and their use at a particular site is then an installation option. We provided two compilation switches, one for inlining (`LNXI`) and another for register allocation (`LNXR`). Each of these switches applies both to GCC, where it controls the output of what we call "linkage information", and to GDB, which uses this linkage information to hide the optimization from the user.

We have used the modified GCC to compile both itself and GDB, under various optimization levels and for three targets. Further, the modified GDB passes the standard test suite. In fact, it succeeds in a place that the test suite was expected to fail.

3 Distribution

We have announced the availability of this system to GCC and GDB interest groups. We are releasing it as standard "patch" files based on the most recent released versions of GCC (2.7.2) and GDB (4.16). It is available under the standard GNU license.

The subsections below are the notes that we distribute with the modifications to GCC and GDB.

3.1 General Information

A modified GCC and GDB hide the effects of inline optimization and of register allocation. The basic idea is that with these modifications, GDB behaves essentially the same whether or not GCC has compiled a program with these optimizations turned on. Details follow on that "essentially the same" means. Keep in mind that there is no change in the way that you use GDB; it just behaves better.

If you already know how these optimizations interfere with debugging, skip this paragraph. The major problem with inlining is that setting breakpoints in an inlined function has no effect when executing inlined instances of the function. A less negative, but potentially confusing, impact is that you see no stack frames corresponding to invocations of the inlined functions---GDB presents only those stack frames that correspond exactly to the physical stack in the "inferior", i.e., the process being debugged. Further, the values of arguments and results of inline functions are available only erratically. The major problem with register allocation is that GDB will lie to you about the value of a variable. This arises because several distinct variables may be assigned to the same register. Ask

for the value of any of the variables and GDB will simply tell you the value in that register, interpreted as having the type of the variable. A further consequence of register allocation is that values defined by the program are destroyed by execution of the program, and are thus unavailable to the GDB user.

GCC now works slightly differently with the `-finline` switch, which in the revised version causes inlining whether or not you have specified `-O`. Thus, it is now possible to perform inlining and no other optimizations. Similarly, GCC now has a new `-fregisters` switch, which causes optimized register allocation whether or not you have specified `-O`. The user may supply any subset of the `-finline` and the `-fregisters` switches. The major pragmatic reason for the new switch behavior is that it enables working on the optimizations in combination with each other and in isolation from other optimizations.

A GDB user may now set a breakpoint in a function that has been inlined, and the program will break in any of the instances where the function has been inlined. Further, when the user asks to view the stack, GDB will supply artificial frames to indicate the source semantics (rather than the target semantics) of function calls. Further, values in these frames will be available in the usual way. Similarly, when a GDB user asks for the value of a variable which is not "current", i.e., some other value is stored in the variable's location, GDB will indicate the non-currentness, either with a "?" when printing the parameter list in a frame or by saying `Value of "... is unavailable here`. In particular, you will get this message when asking for the value of an uninitialized local. Further, GDB will often rescue the values of variables destroyed by execution, thereby making them available to the GDB user, who remains blithely unaware that without these modifications, GDB would like about the values of these variables.

At present, these modifications are implemented only for the "stabs" debugging format (used by BSD systems). Lamentably, they do not work with the `-gstabs` switch available under some systems (e.g., MIPS), because these systems encapsulate stabs in the ECOFF format in a non-extensible way. It would not take much work to extend the encapsulation machinery to handle our extensions to stabs, nor to provide the linkage information under other symbol table formats; volunteers are welcome.

3.2 When to Report a Bug

The behavior of the modified GDB on programs compiled with or without the `-finline` and the `-fregisters` switches is so close that a practical "correctness criterion" for the modifications can be stated in terms of what differences can be observed. Users are encouraged to report violations of the following behavior:

Compile a program with the modified GCC in three ways: with the `-g` switch only (the "non-optimized" version), with the `-g` switch and any non-empty subset of the `-finline` and `-fregisters` switches (the "optimized debugging" version) and with no `-g` switch, and finally with the same non-empty subset of the `-finline` and `-fregisters` switches (the "optimized non-debugging" version).

The machine code for the optimized debugging and optimized non-

debugging versions is identical. (Great care was taken not to modify GCC to change code generation. Users are encouraged to be very diligent in reporting any failures in this regard.)

Use the modified GDB on the non-optimized and optimized debugging versions of the program, in the ordinary way. (There are neither added nor deleted GDB commands.) Then the only detectable difference in the modified GDB's behavior on the two versions of the program arise as follows:

The machine addresses that GDB prints out may differ (because inlining changes the size of code).

The **break** (a.k.a. **b**) command may print out a message **Breakpoint 1 at 0xn, . . .**. The **. . .** means that the inferior may break at one of several pc's, all corresponding to the same place in the source. If the function is uncompiled (because it is an **extern inline** or because it is totally inlined, either because it is a method in some C++ class or by the use of **-O3**), **n** will be 0.

An uncompiled function cannot be used in expressions. (It is possible to remove this restriction, by calling it from an inlined instance, but this is tricky and not worth it now.) Similarly the **disassemble** and **x** commands each give an error for uncompiled functions.

Some variables may print as "?" or cause the "unavailable" error, but only in situations in which the variable has actually been overwritten with another value, and sometimes, not even then (see next item). (If the variable is "dead" but happens not to be overwritten, you will still be able to examine it.) There are still some places where a parameter will disappear entirely from GDB's knowledge, but in all currently known cases, this is due to some optimization other than the two considered, so is outside the purview of the current project.

GDB will rescue the value of an about-to-be-clobbered variable in any function which:

- has a breakpoint, or
- has been arrived at with a **step**, **finish**, or **return** command.

Call such functions "safe". Once a safe function is entered, its variables as well as the variables of all recursive activations will be rescued, even upon removal of the breakpoint which caused the function to be safe.

The idea is that the variables of a function in which a user appears interested are always available (once initialized). Pragmatics prohibit making this guarantee for all the variables of the program. The above

rules seem a reasonable approximation of when a user is interested in a function, and require no conscious effort or new GDB commands. Counter-suggestions are welcome.

The use of machine-level GDB commands, e.g., `stepi`, will, in general, result in different behavior (because a different machine language program is being run). Similarly, "maintenance commands" (e.g., `maint print {p,m} symbols`), which print out internal GDB data, will of course give different results, and `info address` and `info line` behave slightly differently in inline situations. Likewise, the value of `$reg` will vary between the two versions.

The use of `-finline` and `-fregisters` in combination with other optimizations does not have similar guarantee, but at the very least, GDB will not crash.

3.3 For the Experienced User

Supporting the above criterion required modifications to the implementation of many GDB commands. These notes are of interest only to the experienced GDB user, who knows about behavior dependent on these optimizations. A naive user will see only the expected.

`frame`, `up`, `down`: these introduce frames that are in the source semantics but not in the inferior's physical stack; they are also responsible for the "?" indication of a parameter's unavailability.

`break`, `delete`, `clear`, `info b`: for setting, unsetting, and examining breakpoints; the idea is that one source location may be mapped to many pc values.

`step`, `next`, `cont`, `jump`, `until`, `return`, `finish`: controlling execution in the inferior; some of these are cute---for example, GDB may give the appearance of entering or leaving a function without ever running the inferior (because a single pc has an ambiguous interpretation of being just before or just after a function entry or exit).

`list`: for examining source text

`info line`: if the argument is an inline function, it does not give pc information, but indicates its file and line; if the argument specifies a line within an inline function, again, it does not give pc information, but indicates that it is an inline, possibly uncompiled, function. (Well, on this one, maybe a naive user might not expect what happens, but it's pretty self-explanatory.)

info address: indicates an uncompiled function.

3.4 Pragmatics

The additional functionality of GDB comes at some expense in both compilation time and in object file size. The GCC file `final.c`, about 146000 source bytes, has the following costs under the modified and unmodified versions of the compiler:

	modified	original
time in parse	2.840000	2.970000
time in integration	0.000000	0.000000
time in jump	0.790000	0.810000
time in cse	3.660000	3.630000
time in loop	0.260000	0.250000
time in cse2	2.800000	2.710000
time in branch-probabilities	0.000000	0.000000
time in flow	0.490000	0.510000
time in combine	2.640000	2.640000
time in sched	0.790000	0.800000
time in local-alloc	0.690000	0.710000
time in global-alloc	0.660000	0.680000
time in sched2	0.450000	0.510000
time in dbranch	0.890000	0.880000
time in shorten-branch	0.130000	0.120000
time in stack-reg	0.000000	0.000000
time in final	1.160000	0.760000
time in varconst	0.130000	0.060000
time in dump	0.000000	0.000000
time in symout	0.160000	0.120000
total	18.540000	18.160000 (2.0% increase)

The size of the modified compiler's output is 107301 bytes, an increase of 7.3% from the 99933 bytes produced by the original compiler.

We have not done timing studies of GDB. Sometimes it seems slower than unmodified GDB, particularly, perhaps, in functions which have a busy loop (where rescuing is happening) and a break is set outside the loop. But for the most part, performance differences are imperceptible.

Although the intent of this project was to hide optimizations from the GDB user, a fallout is that GDB informs the user when variables values are not available, regardless of the reason for the unavailability. Thus, the modified GDB issues an error to a user who is trying to examine an uninitialized variable (independent of optimization level). This has prevented confusion more than once, and is worth the price of any added GDB expense.

3.5 Caveats

Although every attempt was made to do everything in a target-independent way in both GCC and GDB, I have debugged GCC only on the SPARC, the MIPS (the exercise for the latter target was the first stage in discovering that `-gstabs` encapsulates stabs in ECOFF in a non-extensible way), and the 386, and GDB only on the SPARC and 386. All of the development work was on the SPARC. Porting GCC to the MIPS revealed one existing GCC bug and two from this project; porting it to the 386 (the first non-RISC platform) revealed four minor nits. So porting to a fourth target will probably reveal some problems, but nothing major.

The system has been used on both C and C++ programs, but much more on C than C++. It has not been used at all on Modula-2, Ada, or Fortran. Judging from the incremental effort to get C++ working after C was working (e.g., proper handling of the `this` parameter required a new kind of information to be transmitted from GCC to GDB), other languages will probably not work out-of-the-box, but neither will they require much effort.

While combinations of `-finline` and `-fregisters` have been extensively tested, such testing was not done in combination with other optimizations. On the other hand, this GDB was used to debug GCC compiled with `-O2`, so it is not completely unrobust.

The implementation assumes that a function begins and ends in the same file. This restriction could be dropped, but requires a much more complicated internal data structure, judged not worth it.

Behavior is currently not correct on inline functions with no code at all. This too can be fixed, but in the absence of user requests, seems less important than other items on the agenda. The priority might change if other optimizations frequently optimize away the contents of inline functions.

Interaction with the `catch` command has not been tested.

The following comes from the Inline page under GCC in `M-x info` (cf. The above discussion of the change to the `-finline` switch):

GNU C does not inline any functions when not optimizing. It is not clear whether it is better to inline or not, in this case, but we found that a correct implementation when not optimizing was difficult. So we did the easy thing, and turned it off.

The scary part is that in the experience of this project, a correct implementation when not optimizing was not only easy, but in fact simply worked when tried. So who knows what terrors lie in wait.

Even without trying it, the modifications are guaranteed not to work with threads. The upgrade does not look difficult, but a threads package was not available for use on this project.

3.6 For GCC/GDB Cognoscenti Only

This section gives a brief account of implementation issues.

There are five new kinds of “stab” directives:

N_INLINE: marks an instance of an inlined function, giving its name, where its result is returned (in the string, following a “.”, in the same way that places of locals are specified), and the address at the end of its “prologue” (the instructions that set up its arguments). The immediately following block (balancing **N_LBRAC**/**N_RBRAC** directives) is the block for the inline instance. Formals must appear for each inline instance because, in general, the formals are in different places. Their places of formals appear just before the **N_INLINE** directive.

N_XINLINE: specifies an uncompiled function. The “X” is because the original motivation was to handle `extern inline`, but later it was also found to be required for an inlined C++ method that does not need to be compiled and from use of the `-O3` switch. The entry gives its name, result type, and first and last lines.

N_OK and **N_NOK**: these always follow the stab for a variable, and for a particular variable, alternate. They indicate where the variable becomes available and unavailable. There are two associated values, a delta and a label. If the delta is 0, the [un]availability arises from a control flow merge and begins immediately at the label. Otherwise, the unavailability is caused by an instruction beginning at the label and if the variable becomes available there, is available at or after the label + delta.

N_JUMP: these follow the stab for a function or for a variable. When such a stab follows a variable, it indicates a jump where the variable needs to be rescued (because flow leaves a place where the correct value for a variable is in its location to a place where this guarantee does not hold.) The purpose of these stabs after a function stab is not worth explaining here.

There is also a new "Symbol Descriptor", O, which says that the variable is equal to the current value of the frame pointer plus the value in the stab entry. It is now produced in general, but has been encountered only in connection with `this` parameters, which necessitated its introduction.

GCC modifications:

The `-finline` switch now works as described above.

The `-fregisters` switch was added, and works as described above.

The definition of `tree_block` has three new fields:

`end_prologue`: gives the name of the label at the end of a prologue of an inline function.

`inline_result`: says where an inline result is.

`variables_regions`: encodes where each of the variables of the block is and is not available.

The `inline_flag` of a `tree_decl`, previous unused for variables, now means `ok_on_entry`, i.e., it is true for parameters of a function but not of local variables.

`CODE_LABEL` rtx's have three new fields: `LABEL_U_LIST`, `LABEL_NOK_LIST`, and `LABEL_WAS_OK_LIST`. One of these overlaps with an existing field, so these rtx's are now two words longer. These rtx's also have a new flag, `AFTER_BARRIER`, shared with the `internal` flag.

Most of the modifications to GCC have to do with setting up the above fields and producing revised stab data from them. It was also necessary to be a bit more careful with the data in the `vars` field, because these supply the formals for inlined instances.

GDB modifications:

The `partial_symtab` data structure now has an `inline_names` field to tell it whether it is worth reading the full symtab to find out more about the uses of an inline function (necessary when setting a breakpoint on an inline function).

The block data structure now incorporates the following fields:

`inline_function`: tells whether a block corresponds to an instance of an inline function, and names the function.

`end_prologue`: label after arguments have been evaluated.

`return_aiclass`: address class for the function result.

`return_value`: says where the result is found.

The symbol data structure now has:

`aux_value.fun.{file,line}`: for a function, says where it ends.

`aux_value.fun.rescue_jumps`: records `N_JMP` data for the function.

`aux_value.ok_regions`: for a variable, says where it is available.

The `frame_info` data structure now has an `inline_function` field, indicating the artificial frames mentioned earlier.

Additions to the breakpoint data structure:

`num_inline_breaks` and `inline_breaks`: these provide a list of all of the addresses in addition to the (pre-existing) address of the compiled function. If the function is uncompiled, its address field is -1 (but prints as 0x0).

`sym`: gives the function in which the breakpoint appears.

`frame_inline_depth`: when the frame field is non-null, this is the depth of the conceptual inline frame relative to the physical frame in the inferior.

`watchpoint_frame_inline_depth`: similar to the previous, but with respect to `watchpoint_frame`.

There are two new breakpoint types, `bp_rescue` and `bp_abandon`.

The `enum address_class` type has a new member: `LOC_OFFSET`, corresponding to the `O` symbol descriptor in a stab.

The modifications to GDB are surprisingly large in number, but they all transact with the above additional fields of data structures in more or less obvious ways.

DISTRIBUTION LIST

addresses	number of copies
DOUGLAS A. WHITE RL/C3C3 525 BROOKS RD. ROME, NY 13441-4505	10
SOFTWARE OPTIONS, INC. 22 HILLIARD STREET CAMBRIDGE MA 02138	5
ROME LABORATORY/SUL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
RELIABILITY ANALYSIS CENTER 201 MILL ST. ROME NY 13440-3200	1
ROME LABORATORY/C3AB 525 BROOKS RD ROME NY 13441-4505	1
ATTN: RAYMOND TADROS GIDEP P.O. BOX 3000 CORONA CA 91713-8000	1

AFTT ACADEMIC LIBRARY/LDEE 1
2950 P STREET
AREA 8, BLDG 642
WRIGHT-PATTERSON AFB OH 45433-7765

OL AL HSC/HRG, BLDG. 190 1
2698 G STREET
WRIGHT-PATTERSON AFB OH 45433-7604

US ARMY STRATEGIC DEFENSE COMMAND 1
CSSD-IM-PA
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

COMMANDER, TECHNICAL LIBRARY 1
4747000/C0223
NAVAIRWARCENWPNDIV
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6001

SPACE & NAVAL WARFARE SYSTEMS 2
COMMAND (PMW 178-1)
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200

COMMANDER, SPACE & NAVAL WARFARE 1
SYSTEMS COMMAND (CODE 32)
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200

CDR, US ARMY MISSILE COMMAND 2
RSIC, BLDG. 4484
AMSMI-PD-CS-R, DDOS
REDSTONE ARSENAL AL 35898-5241

ADVISORY GROUP ON ELECTRON DEVICES 1
SUITE 500
1745 JEFFERSON DAVIS HIGHWAY
ARLINGTON VA 22202

REPORT COLLECTION, CIC-14 MS P364 LDS ALAMOS NATIONAL LABORATORY LDS ALAMOS NM 87545	1
AEDC LIBRARY TECHNICAL REPORTS FILE 100 KINDEL DRIVE, SUITE C211 ARNOLD AFB TN 37389-3211	1
COMMANDER USAISC ASHC-IMD-L, BLDG 61801 FT HUACHUCA AZ 85613-5000	1
US DEPT OF TRANSPORTATION LIBRARY FB10A, M-457, RM 930 800 INDEPENDENCE AVE, SW WASH DC 22591	1
AIR WEATHER SERVICE TECHNICAL LIBRARY (FL 4414) 859 BUCHANAN STREET SCOTT AFB IL 62225-5118	1
AFIWC/MSO 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-7016	1
SOFTWARE ENGINEERING INSTITUTE CARNEGIE MELLON UNIVERSITY 4500 FIFTH AVENUE PITTSBURGH PA 15213	1
NSA/CSS K1 FT MEADE MD 20755-6000	1
DCMAD/WICHITA/GKEP SUITE 2-34 401 N MARKET STREET WICHITA KS 67202-2095	1

PHILLIPS LABORATORY
PL/TL (LIBRARY)
5 WRIGHT STREET
HANSCOM AFB MA 01731-3004

1

THE MITRE CORPORATION
ATTN: E. LADURE
D460
202 BURLINGTON RD
BEDFORD MA 01732

1

DOUSD(P)/DTSA/DUTD
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

2

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.